Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

Отчёт по лабораторной работе № 6

Дисциплина: Проектирование мобильных приложений

Тема: Многопоточные приложения.

Выполнил студент гр. 3530901/90201 _____ Е.К. Борисов
(подпись)

Принял старший преподаватель _____ А.Н. Кузнецов
(подпись)

“_____” _____ 2021 г.

Санкт-Петербург
2021

Оглавление

Цели:

Организация обработки длительных операций в background (worker) thread:
Запуск фоновой операции (Coroutine/ExecutionService/Thread)
Остановка фоновой операции (Coroutine/ExecutionService/Thread)
Публикация данных из background (worker) thread в main (ui) thread.

Освоить 3 основные группы API для разработки многопоточных приложений:
Kotlin Coroutines
ExecutionService
Java Threads

Задачи:

- Разработайте несколько альтернативных приложений «не секундомер», отличающихся друг от друга организацией многопоточной работы. Опишите все известные Вам решения.
- Создайте приложение, которое скачивает картинку из интернета и размещает ее в ImaveView в Activity. Используйте ExecutorService для решения этой задачи.
- Перепишите предыдущее приложение с использованием Kotlin Coroutines.
- Скачать изображение при помощи библиотеки на выбор.

Альтернатива “не секундомеру” (Java Threads)

Данное решение использует потоки Threads из Java.

Чтобы создать поток вызывается конструктор, в котором описывается ожидаемое поведение от потока. В большинстве случаев в конструктор передается экземпляр, который наследуют интерфейс Runnable. Этот интерфейс содержит единственный абстрактный метод – void run(), который необходимо переопределить, с него начинается работа нового потока.

При помощи метода run() можно запустить код из Runnable в текущем потоке или же использовать метод start() для запуска в новом потоке.

Чтобы контролировать поток можно воспользоваться методом interrupt(). Данный метод устанавливает статус прерывания у потока. Данный метод не завершает

поток, а только устанавливает статус. Для проверки текущего статуса потока используется метод isInterrupted(). Также, если обрабатывать исключение InterruptedException, то статус потока сбрасывается.

Было переделано приложение “не секундомер” в соответствии с описанными выше высказываниями, код изменений представлен ниже.

Листнг 1 continuewatch (Java Threads)

```
private fun createThread() = Thread {
    try {
        while (!Thread.currentThread().isInterrupted) {
            Log.d("STATE", "${Thread.currentThread()} is iterating")
            Thread.sleep(1000)
            textSecondsElapsed.post {
                secondsElapsed++
                textSecondsElapsed.text = resources.getQuantityString(
                    R.plurals.second_elapsed,
                    secondsElapsed,
                    secondsElapsed
                )

            }
        }
    } catch (e: InterruptedException) {
        Log.d("STATE", "${Thread.currentThread()} catch exception")
    }
}

override fun onStart() {
    super.onStart()
    Log.d("STATE", "onStart")

    backgroundThread = createThread()
    backgroundThread.start()
}

override fun onStop() {
    super.onStop()
    Log.d("STATE", "onStop")

    backgroundThread.interrupt()
}
```

Код создания потока теперь обернуть конструкцией try-catch, если будет получено исключение InterruptedException, например, при сворачивании приложения, в Logcat будет выведено сообщение о том, что было поймано исключение и поток прекратит свою работу.

Для создания нового потока вызывается соответствующая функция и затем запускается при помощи метода start(), то есть мы каждый раз создает новый поток а не продолжаем работу прерванного потока. А для его прекращения используем метод interrupt(). Прекращение обусловлено реализацией программы.

```
2021-11-20 09:51:32.640 18656-18700/com.example.continuewatch D/STATE: Thread[Thread-3,5,main] is iterating
2021-11-20 09:51:33.643 18656-18700/com.example.continuewatch D/STATE: Thread[Thread-3,5,main] is iterating
2021-11-20 09:51:34.334 18656-18656/com.example.continuewatch D/STATE: onStop
2021-11-20 09:51:34.334 18656-18700/com.example.continuewatch D/STATE: Thread[Thread-3,5,main] catch exceptio
2021-11-20 09:51:53.991 18656-18656/com.example.continuewatch D/STATE: onStart
2021-11-20 09:51:53.995 18656-18712/com.example.continuewatch D/STATE: Thread[Thread-4,5,main] is iterating
```

Рис. 1 Тестирование с помощью Logcat.

Можно заметить, что работает только один фоновый поток секундомера.

Также, так как при создании нового потока отсчитывается ровно одна секунда, а прекращение потока могло пройти, когда было, к примеру, 1 секунда и 800 миллисекунд, начало отсчета с 0 миллисекунд будет некорректным, было придумано следующее решение данной проблемы:

Листнг 1 Исправление подсчета секунд (Java Threads)

```
private var isLeft = false
private var milliFirst: Long = 0
private var milliSecond: Long = 0


private fun createThread() = Thread {
    try {
        while (!Thread.currentThread().isInterrupted) {
            Log.d("STATE", "${Thread.currentThread()} is iterating")
            Log.d("STATE", "Time before onPause: ${System.currentTimeMillis()}")
            if (!isLeft) {
                milliFirst = System.currentTimeMillis()
                Thread.sleep(1000)
            } else {
                Log.d("STATE", "Delay: ${(milliSecond-milliFirst).toString()}")
                isLeft = false;
                Thread.sleep(1000 - (milliSecond - milliFirst))
            }
            textSecondsElapsed.post {
                secondsElapsed++
                textSecondsElapsed.text = resources.getQuantityString(
                    R.plurals.second_elapsed,
                    secondsElapsed,
                    secondsElapsed
                )
```

```
                }
            }
        } catch (e: InterruptedException) {
            Log.d("STATE", "${Thread.currentThread()} catch exception")
        }
    }

    override fun onPause() {
        super.onPause()
        isLeft = true
        Log.d("STATE", "Time in onPause: ${System.currentTimeMillis()}")
        milliSecond = System.currentTimeMillis()
    }
```

Теперь мы два раза запоминает системное время, каждый раз на каждой итерации цикла и в методе onPause, а также устанавливаем флаг isLeft в true, для того чтобы отслеживать, было ли прервано приложение, таким образом, опять же, например при сворачивании приложения, у нас будет время начала отсчета и время, когда был вызван метод onPause, и при разворачивании приложения мы просто отнимаем в Thread.sleep разницу между этими двумя точками времени.

```
2021-11-20 10:18:00.894 19117-19145/com.example.continuewatch D/STATE: Thread[Thread-2,5,main] is iterating
2021-11-20 10:18:00.894 19117-19145/com.example.continuewatch D/STATE: Time before onPause: 1637392680894
2021-11-20 10:18:01.895 19117-19145/com.example.continuewatch D/STATE: Thread[Thread-2,5,main] is iterating
2021-11-20 10:18:01.897 19117-19145/com.example.continuewatch D/STATE: Time before onPause: 1637392681897
2021-11-20 10:18:02.481 19117-19117/com.example.continuewatch D/STATE: Time in onPause: 1637392682481
2021-11-20 10:18:02.481 19117-19117/com.example.continuewatch D/STATE: onStop
2021-11-20 10:18:02.482 19117-19145/com.example.continuewatch D/STATE: Thread[Thread-2,5,main] catch exception
2021-11-20 10:18:12.213 19117-19117/com.example.continuewatch D/STATE: onStart
2021-11-20 10:18:12.215 19117-19161/com.example.continuewatch D/STATE: Thread[Thread-3,5,main] is iterating
2021-11-20 10:18:12.215 19117-19161/com.example.continuewatch D/STATE: Time before onPause: 1637392692215
2021-11-20 10:18:12.215 19117-19161/com.example.continuewatch D/STATE: Delay: 584
2021-11-20 10:18:12.268 2511-2511/com.google.android.googlequicksearchbox E/HwDetectorWithState: a: 3
2021-11-20 10:18:12.632 19117-19161/com.example.continuewatch D/STATE: Thread[Thread-3,5,main] is iterating
2021-11-20 10:18:12.635 19117-19161/com.example.continuewatch D/STATE: Time before onPause: 1637392692635
2021-11-20 10:18:13.640 19117-19161/com.example.continuewatch D/STATE: Thread[Thread-3,5,main] is iterating
2021-11-20 10:18:13.641 19117-19161/com.example.continuewatch D/STATE: Time before onPause: 1637392693641
```

Рис. 2 Тестирование подсчета времени.

Как видно по консоли, при сворачивании приложение и последующем его разворачивании следующая секунда наступит через 1000 - 584 секунды.

Альтернатива “не секундомеру” (ExecutorService)

ExecutorService – это JDK API, который позволяет более гибко работать с потоками он имеет удобный механизм контроля асинхронных потоков. Для начала нужно создать объект ExecutorService, в котором использовать класс ThreadPoolExecutor.

Execute(Runnable) – запускает переданный Runnable асинхронно.

Submit(Runnable) – запускает поток и возвращает экземпляр класса Future, который можно использовать для различных действий над потоком.

Метод get() объекта future можно заблокировать текущий поток, тем самым он будет ждать пока фоновый поток не завершится. Помимо этого, возможно задать время, в течение которого поток должен завершиться, чтобы понимать, что он точно будет завершен.

Метод cancel() объекта future останавливает работу потока, если передать ему в качестве аргумента True значение.

Для общения между потоками также вместо Runnable можно использовать Callable объект. В отличие от Runnable, тут мы должны переопределить метод call, который возвращает результат, заданный типом-параметром.

Код переделанной версии программы представлен ниже

Листинг 3 continuewatch (ExecutorService)

```
private val pool : ExecutorService = Executors.newSingleThreadExecutor()
var secondsElapsed: Int = 0
lateinit var textSecondsElapsed: TextView
private var isLeft = false
private var milliFirst: Long = 0
private var milliSecond: Long = 0

private lateinit var backgroundFuture: Future<*>

private fun submitThread(executorService: ExecutorService) =
executorService.submit {
    while (true) {
        Log.d("STATE", "${Thread.currentThread()} is iterating")
        if (!isLeft) {
            milliFirst = System.currentTimeMillis()
            Thread.sleep(1000)
        } else {
            Log.d("STATE", "Delay: ${(milliSecond-milliFirst).toString()}")
            isLeft = false;
            Thread.sleep(1000 - (milliSecond - milliFirst))
        }
        textSecondsElapsed.post {
            secondsElapsed++
```

```
                textSecondsElapsed.text = resources.getQuantityString(
                    R.plurals.second_elapsed,
                    secondsElapsed,
                    secondsElapsed
                )

            }
        }
    }

override fun onPause() {
    super.onPause()
    isLeft = true
    Log.d("STATE", "Time in onPause: ${System.currentTimeMillis()}")
    milliSecond = System.currentTimeMillis()
}

override fun onStart() {
    super.onStart()
    backgroundFuture = submitThread(pool)
}

override fun onStop() {
    super.onStop()
    backgroundFuture.cancel(true)
}
```

Для начала мы создаем пул с одним потоком, далее в onStart при помощи метода submit, описанном в функции submitThread, поток добавляется в очередь на исполнение и объект с помощью возвращённого объекта future, с помощью которого можно контролировать поведение потока, мы прерываем поток при помощи метода cancel(), таким образом все потоки будут выполнятся в одном пуле, это можно увидеть на рисунке ниже.

```
2021-11-20 11:30:49.497 20652-20711/com.example.continuewatch D/STATE: Thread[pool-2-thread-1,5,main] is iterating
2021-11-20 11:30:50.500 20652-20711/com.example.continuewatch D/STATE: Thread[pool-2-thread-1,5,main] is iterating
2021-11-20 11:30:51.035 20652-20652/com.example.continuewatch D/STATE: Time in onPause: 1637397051035
2021-11-20 11:30:54.134 20652-20711/com.example.continuewatch D/STATE: Thread[pool-2-thread-1,5,main] is iterating
2021-11-20 11:30:54.134 20652-20711/com.example.continuewatch D/STATE: Delay: 535
2021-11-20 11:30:54.183 2511-2511/com.google.android.googlequicksearchbox E/HwDetectorWithState: a: 3
2021-11-20 11:30:54.600 20652-20711/com.example.continuewatch D/STATE: Thread[pool-2-thread-1,5,main] is iterating
2021-11-20 11:30:55.603 20652-20711/com.example.continuewatch D/STATE: Thread[pool-2-thread-1,5,main] is iterating
```

Рис. 3 Тестирование приложения (ExecutorService)

Альтернатива “не секундомеру” (Coroutine)

Корутины – это альтернатива потокам в языке программирования Kotlin. Однако между корутинами и потоками нет прямого соответствия. Корутина не привязана к конкретному потоку. Она может быть приостановить выполнение в одном потоке, а возобновить выполнение в другом. Корутины являются легковесными и безопасными по сравнению с потоками в Java.

Для использования корутин в андройд разработке нужно подключить библиотеку, для создания корутин есть несколько конструкторов:

Launch – он создает корутину с функционалом описанном внутри блока и запускает эту корутину параллельно с остальным кодом.

Async - запускает отдельную корутину, которая выполняется параллельно с остальными корутинами. Кроме того, она возвращает объект Deferred, который ожидает получения результата корутины. Для получения результата из объекта Deferred применяется функция await().

RunBlocking - Основной поток, вызывающий runBlocking, блокируется до завершения сопрограммы внутри runBlocking.

Также необходим scope, который является некой связью между кодом который выполняется параллельно с кодом корутины. Конструкция launch возвращает экземпляр класса Job. С помощью join(), объект Job может быть присоединен к другому потоку.

Метод cancel() объекта Job позволяет остановить поток, а для приостановления работы корутины на определенное количество времени используется функция delay().

При помощи lifeCycleScope можно реализовать выполнение корутины таким образом, чтобы оно происходило, когда приложение находится на экране пользователя (Resumed), код изменой версии программы представлен ниже.

Листинг 4 continuewatch (Coroutines)

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    textSecondsElapsed = findViewById(R.id.textSecondsElapsed)
```

```
    val button: Button = findViewById(R.id.button)
    button.setOnClickListener {
        val intent = Intent(this, MainActivity2::class.java)
        startActivity(intent)
    }

    lifecycleScope.launchWhenResumed {
        while (isActive) {
            Log.d("state", "Coroutine is iterating")
            if (!isLeft) {
                milliFirst = System.currentTimeMillis()
                delay(1000)
            } else {
                Log.d("STATE", "Delay: ${(milliSecond-milliFirst).toString()}")
                isLeft = false;
                delay(1000 - (milliSecond - milliFirst))
            }
            textSecondsElapsed.post {
                secondsElapsed++
                textSecondsElapsed.text = resources.getQuantityString(
                    R.plurals.second_elapsed,
                    secondsElapsed,
                    secondsElapsed
                )

            }
        }
    }
}
```

```
2021-11-20 10:51:39.538 20249-20249/com.example.continuewatch D/state: Coroutine is iterating
2021-11-20 10:51:40.539 20249-20249/com.example.continuewatch D/state: Coroutine is iterating
2021-11-20 10:51:41.543 20249-20249/com.example.continuewatch D/state: Coroutine is iterating
2021-11-20 10:51:42.186 20249-20249/com.example.continuewatch D/STATE: Time in onPause: 1637394702185
2021-11-20 10:51:42.247 20249-20249/com.example.continuewatch D/STATE: onStop
2021-11-20 10:51:43.480 20249-20249/com.example.continuewatch D/STATE: onStart
2021-11-20 10:51:43.481 20249-20249/com.example.continuewatch D/state: Coroutine is iterating
2021-11-20 10:51:43.481 20249-20249/com.example.continuewatch D/STATE: Delay: 645
2021-11-20 10:51:43.519 2511-2511/com.google.android.googlequicksearchbox E/HwDetectorWithState: a: 3
2021-11-20 10:51:43.837 20249-20249/com.example.continuewatch D/state: Coroutine is iterating
2021-11-20 10:51:44.839 20249-20249/com.example.continuewatch D/state: Coroutine is iterating
2021-11-20 10:51:45.840 20249-20249/com.example.continuewatch D/state: Coroutine is iterating
2021-11-20 10:51:46.843 20249-20249/com.example.continuewatch D/state: Coroutine is iterating
```

Рис. 4 Тестирование приложения (Coroutines)

Загрузка изображения (Потоки)

Загрузим изображение по ссылке используя потоки и поместим его в imageView, если загружать картинки в UI потоке, то будет получена ошибка поэтому загрузим картинку в отдельном потоке при помощи ExecutorService во ViewModel.

Листинг 5 ViewModelDownload.kt

```
class ViewModelDownload: ViewModel() {
    val bitmap: MutableLiveData<Bitmap> = MutableLiveData()

    private val downloadThread : ExecutorService =
Executors.newSingleThreadExecutor()

    fun downloadImage(url: URL) {
        downloadThread.execute {
            Thread.sleep(3000)

bitmap.postValue(BitmapFactory.decodeStream(url.openConnection().getInputStream(
)))
        }
    }

}
```

Листинг 6 MainActivity.kt

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)
    val mainViewModel: ViewModelDownload by viewModels()
    binding.btnDownload.setOnClickListener {

mainViewModel.downloadImage(URL("https://yapcdn.net/se4/32/rlgjPOGyJuwTek.png"))
    }
    mainViewModel.bitmap.observe(this) {
        binding.image.setImageBitmap(it)
    }

}
```

Загружаем изображение в отдельном потоке с помощью bitmap изображение, а затем устанавливаем обработчик событий на кнопку загрузки изображения, по нажатию которой будет начинать загрузка и по окончанию она появляется в imageView. Также для корректной работы приложения нужно разрешить доступ к интернету в манифесте добавлением следующих строк:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Загрузка изображения (Coroutines)

Изменим предыдущее решение, теперь используя корутины

Листинг 7 ViewModelDownload.kt (Coroutines)

```
class ViewModelDownload : ViewModel() {

    val bitmap: MutableLiveData<Bitmap> = MutableLiveData()

    fun downloadImage(url: URL) {
        viewModelScope.launch(Dispatchers.IO) {
            delay(3000)
            val pic = BitmapFactory.decodeStream(
                url.openConnection().getInputStream()
            )
            withContext(Dispatchers.Main) {
                bitmap.value = pic
            }
        }
```

В отличие от предыдущего решения здесь мы используем обычное присвоение, благодаря переключению потоков, вместо postValue.

Загрузка изображения (Библиотеки)

При помощи библиотеки Picasso, мы можем загрузить изображение в одну строчку следующим образом:

Листинг 8 MainActivity.kt (Picasso)

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)
    binding.btnDownload.setOnClickListener {
        Picasso.get().load(
            "https://yapcdn.net/se4/32/rlgjPOGyJuwTek.png"
        ).into(binding.image);
    }

}
```

Ответы на вопросы

Threads (Java)

Поток в Java представлен в виде экземпляра класса java.lang.Thread.

- Создание потока:

Происходит с помощью конструкторов, в конструктор передается экземляр, который наследует интерфейс Runnable.

- Запуск потока:

Запуск происходит при помощи метода start()

- Завершение потока

Происходит при помощи установки статуса прерванности с помощью метода Thread.interrupt(). Можно получить статус этого флага при помощи isInterrupted(). Некоторые объекты могут выдавать исключения из-за статуса, например, как было показано в первом пункте работы Object.wait(), чаще всего это бывает InterruptedExceoption, вследствие этого нужно бывает обрабатывать реакцию программы на прерывание. Для остановки потока возможно использовать также метод stop(), но его использование очень не рекомендуется, так как ведет к ошибкам вследствие того, что трудно предсказать конечное состояние объектов

- Передача в UI поток

Происходит при помощи Activity.runOnUiThread(Runnable), View.post(Runnable), View.postDelayed(Runnable, long).

ExecutorService:

- Создание:

Для начала нужно создать объект ExecutorService, в котором использовать класс ThreadPoolExecutor/ScheduledThreadPoolExecutor тем самым создав пул потоков.

При создании можно указывать различные параметры, данная фитча дает большое преймущество по сравнению с потоками в Java.

- Запуск

Execute(Runnable) – запускает переданный Runnable асинхронно.
Submit(Runnable) – запускает поток и возвращает экземпляр класса Future, который является отображением асинхронных вычислений, которыми можно управлять, используя методы данного класса.

Количество потоков указывается при создании объекта ExecutorService, при запуске новых Runnable, если нет свободных потоков, они добавляются в очередь.

- Завершение

В случае запуска при помощи Execute(Runnable), есть метод shutdown, который завершает запуск новых Runnable и обрабатывает оставшиеся потоки, при добавлении новой задачи будет выдано RejectedExecutuinException, чтобы остановить исполнение потоков, не дожидаясь их обработки используется метод shutdownNow(), однако так может быть получено исключение InterruptedException.

Также у экземпляра класса Future, который возвращает Submit(Runnable) есть метод cancel(), который останавливает заданный поток.

- Передача в UI поток

Происходит при помощи Activity. runOnUiThread(Runnable), View.post(Runnable), View.postDelayed(Runnable, long).

Корутины (Kotlin)

- Создание:

Происходит при помощи конструкторов launch, async, runBlocking. Для выполнения корутин необходимо определить контекст, так как корутины могут выполняться только в контексте корутин.

- Запуск

Происходит при помощи метода launch, который асинхронно запускает корутину. Launch возвращает экземляр класса Job, который используется для контроля над корутиной при помощи соответственных методов. Также в параметрах launch можно указать один из нескольких видов диспетчера корутины, каждый из которых подходит для своих определенных задач.

- Завершение

Происходит при помощи метода cancel() класса Job, или же с помощью жизненных циклов андройд, как было показано в третьем пункте лабораторной работы.

- Передача в UI поток

Происходит при помощи Activity. runOnUiThread(Runnable), View.post(Runnable), View.postDelayed(Runnable, long), а также при помощи переключения контекстов.

Вывод

В данной лабораторной работе были изучены многопоточные андройд приложения. Были изучены разные способы работы с потоками в андройд, их создание запуск и завершение, а также отправка значений в UI поток. Также были получены навыки работы с bitmap благодаря пункту с загрузкой изображения из интернета.

Потраченное время

1,2,3 пункты – заняли в сумме часов 9, точно не засекал выполнение каждого пункта

3,4 пункты ~ 90 минут